Proxies, Application Interfaces, and Distributed Systems

Amitabh Dave, Mohlalefi Sefika and Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61820

Abstract

Proxy objects are local representatives of remote objects in a distributed system. We use proxies to construct a transparent application programming interface (API) for the Choices *distributed operating system. In earlier work, proxies were used in* Choices *to provide a protected, object-oriented interface to system objects. The addition of RemoteProxies allows applications to access all resources in a uniform way by simply invoking methods on objects, irrespective of whether they are local, in the kernel, in a different user virtual address space or remote. We also extend proxies as defined by Shapiro[10] to optimize access to remote and protected objects and to provide support for changing server interfaces. We describe a new remote procedure call (RPC) facility for invoking methods on remote objects through the proxy mechanism. The API is made dynamically reconfigurable by using table lookup to perform all functions normally provided by stubs in conventional RPC implementations[11]. Last, the API permits new versions of a service to be introduced without requiring recompilation of application client code.*

1 Introduction

An application programming interface (API) provides a set of run time facilities and services that may be invoked by applications. In a distributed system, these facilities and services are implemented by functions and servers in the kernel, in separate virtual memory spaces, or on remote computers. A distributed operating system provides location transparent access to remote servers and resources. In this paper, we argue for a transparent application interface that hides whether facilities and services are provided as a library to the application or provided by the distributed system upon which it runs.

In *Choices*, we view all entities in a system as objects that belong to a class hierarchy. The application interface is the mechanism through which application objects invoke facilities and services from other server and peer objects that may be in the kernel, on the local system but in a different virtual memory space, or on a remote system. The object's methods may be accessed independently of these attributes through the transparent API. Unlike conventional APIs which distinguish between a basic set of services and the services provided by servers, the *Choices* API permits the set of functions and services that are available to an application to be dynamic and uniformly accessible.

Proxy objects are local representatives of remote objects in a distributed system[10]. The *Choices* API mechanism is built using proxy objects[2, 9]. In distributed *Choices*, proxy objects represent other objects that cannot be accessed directly from an application. Proxy objects provide indirection and late, run time binding. A method invocation on a proxy results in a corresponding method invocation on the object it represents.

A transparent API for an object-oriented distributed operating system has several benefits. Applications may be written and compiled to be independent of the location of the servers and other resources they access. This simplifies porting and object migration. The API separates the provision of resources from how those resources are implemented. The resources may be implemented in the same user space as the application, in a different user space, in the kernel, or on a remote machine. The user of an application or an operating system load balancing policy may choose an appropriate implementation of a service or function depending upon load, data set size, application host hardware, or other considerations. The application may be run on a "microkernel" with most resources accessed through user level servers or on a more traditional kernel with the resources provided as kernel services.

The use of proxy objects provides a transparent application interface and a simple and uniform access

0-8186-3015-9/92 $03.00 © 1992 IEEE

model for all objects in the system. In *Choices*, proxy objects support the object-oriented notions of class hierarchy and inheritance. Proxy objects may be used to configure dynamically and efficiently the run time support for an application. In addition, proxy objects allow interfaces to be reused when new versions of the classes are introduced. To support dynamic configuration and class versions, we have built a new implementation of the *Choices* proxy object that provides server interfaces with a table driven RPC facility instead of subroutine stubs. This mechanism and how it allows proxy objects to be used to support an open systems object-oriented API are discussed below.

Existing systems provide transparent application interfaces in several ways. For example, in message passing systems, ports are location independant communication entities that represent servers. Restricting access to all system resources through these ports makes the interface transparent[4]. However, applications must distinguish between those resources that can be reached directly and those that can be accessed through the message passing system. Objects of the same class that reside on different remote systems must be accessed using different ports. Remote procedure calls (RPCs) go one step further and hide the message passing system. RPCs have been shown to work well both for cross-machine calls as well as cross-domain calls on the same machine[1]. Current implementations of RPCs force applications to link with libraries of stub functions which implement the actual remote calls. This makes it difficult to change server interfaces dynamically.

The RPC stubs are usually built from the definitions of the procedures. Using RPC support to call methods on objects is simple if all the objects of a given class are remote. However, complications arise when some objects of a class may be local, others remote, and some may be in the kernel. Typically, RPC's employ only one mechanism to marshall a procedure call parameters into a message. Using the same RPC mechanism to marshall a method call to each of these different locations imposes unnecessary overhead. In contrast, our proposed implementation is both transparent and reconfigurable on the fly.

Proxies are the key to our API provisions. Shapiro[10] proposed proxies, which act as the local representatives for groups of distributed servers. We interpret a proxy in more general terms as the local representive of any object which exists in a different address space. Enhancements may optimise access to remote objects, provide support for changing server interfaces, and allow instrumentation of server performance.

The object model in distributed *Choices* is introduced in Section 2. We then describe the existing *Choices* API in Section 3. Next, in Section 4 we list properties of proxies which are useful in API extensions. The table driven RPC scheme based on proxies is then described in Section 5. Related systems are surveyed in section 6. Section 7 highlights our experience and discusses future work.

2 The object model

In *Choices* all entities including files, disk partitions, address spaces (domains), processes, CPUs, messages, servers, classes, semaphores and timers are objects. Objects vary in granularity from fine grain (like messages) to coarse grain (domains). Objects may be located anywhere in the system and may move. Objects communicate with each other by method invocation. The basic aim of the object model is to provide a uniform, object-oriented way of structuring the system as well as applications and allow transparent access to all objects.

All objects are instances of a class belonging to a class hierarchy. Objects can be private or proxiable. Private objects are accessible only within a domain. Proxiable objects may be accessed from other domains, subject to protection policies and constraints. An object is designated as private or proxiable depending its class. Such a class definition is annotated to indicate the methods that may be invoked globally. A further annotation permits processes in other domains to create new instances of the class. References to proxiable objects are made available to other domains by registering the objects with an appropriate *NameServer*.

Method invocation on objects in *Choices* is uniform, irrespective of object location. The distribution of objects is transparent. A reference to a proxiable object is obtained from a *NameServer*. This reference is in the form of a proxy object that represents the remote object. A method call on the proxy object is transformed into a method call on the proxiable object by the proxy object. Figure 1 shows an application program accessing the *NameServer* to find an object using the method lookup on the *NameServer*. The *NameServer* locates and authenticates access to the object and then allocates a proxy object to provide a local representative of the proxiable object.

The new operator may be used to create a new object of a proxiable class. The *ObjectServer* is responsible for object creation, placement and registration of

the object with the *NameServer*. A simplified view of the creation process is shown in Figure 2.

NameServers and *ObjectServers* are proxiable objects. Standard *NameServers* are accessed using default proxy objects that are created when a domain is created. Thus, applications can access the Standard *NameServers* without requiring a lookup.

3 Proxy objects and the kernel

Proxy objects were introduced into *Choices* to provide applications with a protected, object–oriented interface to system objects[10]. The *ObjectProxy* class defines proxy objects. Before an application can invoke a method on a system object, it must first obtain a proxy object for that object from the name server. A method call to an *ObjectProxy* results in a method call to the corresponding system object. The *ObjectProxy* implements an indirection that is used to provide controlled access to system objects from untrusted applications. Besides providing this form of access to existing kernel objects, new kernel objects may be created using the C++ new operator. Dykstra shows how the proxy interface can be used to extend a *Choices* class hierarchy by subclassing kernel classes in user space[2].

3.1 Proxy objects

Applications invoke a method on a kernel object by invoking the corresponding method on its *ObjectProxy*[9]. Messages sent to an *ObjectProxy* are forwarded to the corresponding kernel object. *ObjectProxys* are dynamically created by the kernel during name server lookup of the kernel object and are protected by allocating them in application read-only memory. An *ObjectProxy* shares a virtual function table with other similar proxy objects. Each method of the proxy object represents a method of the kernel object for which it is a proxy. The virtual function table entries for an *ObjectProxy* are identical and reference code that forwards the method call to the kernel object within the kernel.

When invoked, an *ObjectProxy* method traps into the kernel. The trap handler has access to special tables which contain information about each kernel method and its parameters. The trap handler checks the parameters, makes copies of the parameters as necessary, and calls the actual kernel object methods. If proxy objects are passed as parameters into the kernel, the parameter mechanism checks whether they are valid parameters and replaces the proxy object reference with a reference to the actual kernel object. When the method call returns, any results are copied back into the address space of the application. Any pointers to kernel objects that are returned are converted into pointers to proxy objects.

3.2 Specifying the application interface for kernel objects

The tool Proxify++[2] generates the proxy tables that are used by *ObjectProxies*. It also generates the kernel object header files that are used by applications to call kernel objects. The source code declarations of kernel object methods that are to be exported to applications through the API are marked with the keyword **proxiable** (which is an extension to the C++ language.) Proxify++ is a preprocessor that parses the source C++ header files of the kernel objects and compiles *proxiable* methods into:

- *ProxyTables* that are used in the implementation of proxy calls and
- application header files that are linked with applications that want access to the proxied methods, constructors, and classes of kernel objects.

The application interface is generated automatically by running all *Choices* header files through Proxify++[1].

4 Proxies and distributed systems

Shapiro introduced proxy objects as local representatives of remote services in a distributed system[10]. Proxies in *Choices* are local representatives of objects located outside the address space of an application. They are allocated dynamically when an application "looks up" the address of a proxiable object that has been registered with or bound to the nameserver. In this scheme, the internal state of a proxy object records whether a remote or kernel access mechanism is to be used to access a particular object. The parameters of this mechanism can be changed dynamically to allow objects to move while they are used. According to the proxy principle[10], the proxy is the only visible interface to a service. To the application, interacting with the proxy is identical to interacting with the actual object as if it were a local object. Thus proxies

[1] The preprocessor is based on a *MPL* compiler built at the University of Virginia by Ed Loyot and Andrew Grimshaw, which has been extended to recognize the new keyword *proxiable*.

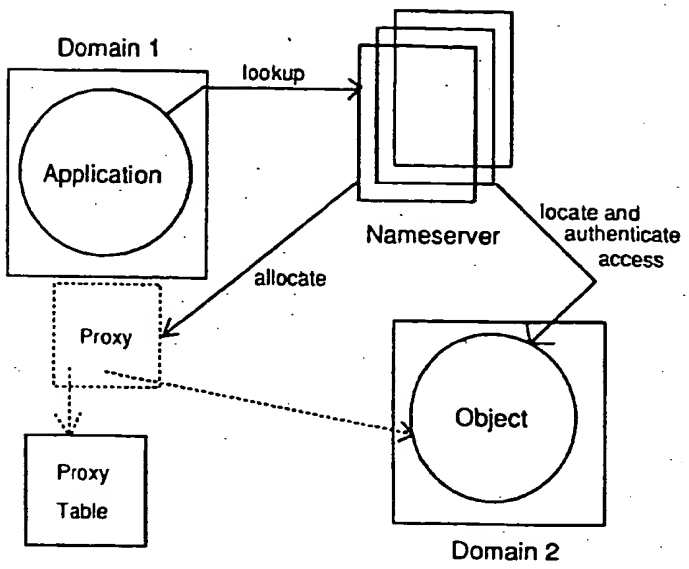

Figure 1: Object lookup

allow transparent access to objects located anywhere in a distributed system.

Our design for distributed *Choices* makes extensive use of proxies. In the following discussion, the term *client* refers to an application that invokes a method. The term *server* refers to an object upon which the method is being invoked. We extend the basic services proposed for proxies in [10] to include dynamic subtyping, multiple versions of server interfaces, instrumentation of client invocations of server methods, and local client caching of remote server information. We now give a more detailed description of the functions provided by a proxy.

Proxies provide three different categories of functions: essential, server specific optimizations and additional.

4.1 Essential functions

These functions are provided by every proxy. Although the essential functions are identical for all proxies, the implementation of the functions depends on server type and location. For example, different parameter marshalling schemes are used for local kernel services and for remote services.

Communication with server : This function is responsible for transferring data between the proxy and the server. Depending on the location of the server, this could involve copying data, mapping address spaces to allow data sharing, or passing messages over the network. Since the location of the server may change after the proxy is allocated in a system implementing object migration, the ability to reconfigure this service dynamically is essential.

Forwarding and marshalling of requests : A request is an invocation of a server method by the client. A proxy, as the server's representative, validates the request by checking parameters. It also marshalls the parameters into a request packet, which is forwarded to the server. Machine dependent translations like byte ordering are also handled here.

Protection : Proxies provide protection to both the client and the server. The client must be assured that the proxy really represents the service it asked for. The assurance is given by having a trusted process allocate the proxy in a separate unmodifiable address space. One way to do this is to use a kernel process and to allocate proxies in read-only memory. The access rights of clients to invoke methods on the object are validated at "look up" time by the nameserver and

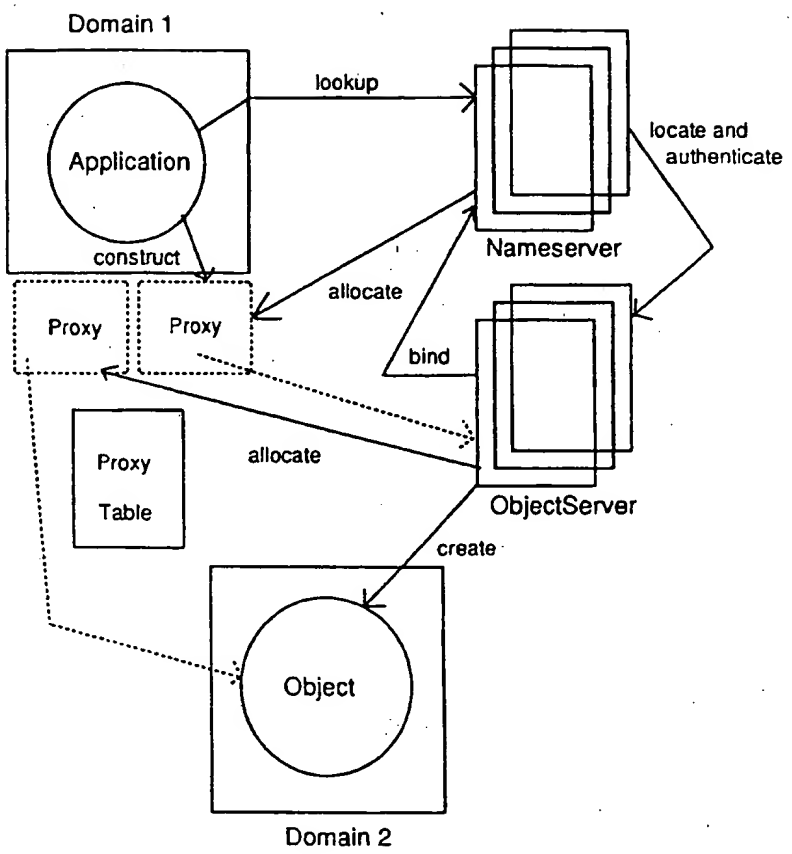

Figure 2: Object creation

again at request time. Restricting the client's access to a subset of the server interface could be implemented by run time checks during each access.

Failures : Failure of a client or server is just as difficult (or easy) to recover from as client or server failures in RPC schemes. Depending on whether just the client or the client machine has failed, the failure can be detected by the proxy or the server. In either case, the server knows of the failure and can handle it appropriately. Server failure is harder to handle because transparency requirements are violated. Aborting the client with a special error message is one option. In any case, the proxy object provides an agent for attempting to implement failure detection and/or recovery as transparently as possible.

4.2 Server specific optimizations

Optimizations can improve the performance of a method call on a proxy object but may make reliability more expensive. In general, a proxy object can record the parameters of a method call and its results and reuse the data to improve performance of similar future method calls. More specific optimizations include caching file system access requests using block or whole file caching and building a hash table of names requested from name servers. Because the application interface is a proxy object, it is easy to provide the object with internal state that records cached data. The proxy object may also be used to support cache coherency and data consistency protocols. Name server proxies may have buffers to store

hints. File server proxies may include cache coherence protocols for shared access to files.

Providing local service : Some requests to remote objects may be more easily provided locally by the proxy itself. For example, to find out more information about the class of a proxied service, the classOf method, which returns the name of the class of an object, could be implemented by the local proxy object.

4.3 Additional services

These services are required only occasionally, for example, when measurements on server utilization by a particular client are being made or when server interfaces have changed.

Performance measurements : Measurements of server response time and its dependence on client location with respect to the server can be made by inserting instrumentation code in the proxy. Adding a counter instrument and a timer instrument to the proxy allows, for example, the measurement of the number of requests to the server from an application and the average response time.

Server versions : Server interfaces may change, either by the addition of more public methods or by altering the type or number of parameters to a method. In the latter case, clients that use the old interface must be recompiled to use the new interface. An alternative approach is to build proxy objects for both interfaces, the proxy object for the old interface converts method calls to the new interface. Default values of missing parameters and type conversions between old and new types can be specified. Although there are limitations to the translations which are feasible, the proxy object removes the need to recompile applications because of minor server changes.

5 Table–driven RPC

In this section we describe the construction of a proxy object mechanism that uses table–driven RPCs to provide applications with transparent system–wide access to objects. We briefly overview different RPC implementations and the limitations of those implementations to support a transparent dynamically–configurable API. Then, we outline our approach.

5.1 RPC implementations

Conventional RPC implementations use stubs to implement non-local calls. A stub generator parses interface specifications for servers and generates stubs which are linked with the applications. A stub implements a remote procedure call by performing parameter format conversions, marshalling the call and parameters into a message, and sending the message to the appropriate server. The functions performed by the stub are determined at compile time and, once linked to an application, the application becomes bound to these functions. The stub approach has the following limitations:

1. Clients can only invoke RPCs on new servers that are subclasses of the class for which the stub was generated.
2. The marshalling, format conversion, and any parameter checking performed by the stub cannot be changed without relinking the client. In particular, this becomes a problem for objects with long lifetimes.
3. Should the client or server move during execution, perhaps as a result of dynamic load balancing or node failure, the stub cannot be altered dynamically to improve performance or recover from the failure.
4. For one client, different stubs must be used to implement RPCs to different objects of the same class if the nature of that access if different. For example, the stub may perform caching optimizations, provide access to kernel objects, local objects in a different virtual memory space or remote objects.
5. Should instances of a new version of the class of a server replace old instances, new stubs must be generated and the clients must be relinked.

5.2 *Choices* proxy RPC

We make use of the existing *Choices* implementation of controlled access to kernel objects and extend the *prozy* mechanism to provide transparent access to remote objects. Instead of including all marshalling and processing code in RPC stubs, we use tables to store information about the appropriate marshalling routines for each parameter in each remote method. These dynamic tables define the interface to each server. A table entry is associated with each object that is accessible to a client application and the

index of the entry is stored in the corresponding proxy object. The proxy object uses its index and these tables to identify the appropriate parameter marshalling functions dynamically. The table-driven approach has the following advantages:

- The tables become a standard interface for the RPC mechanism. They are independent of the specification language (for example, they do not depend on C++ or Proxify++.) Different RPC parameter specification languages can be used for different languages without having to change the table format.
- The tables can be updated dynamically. This makes it possible to add new class entries as needed.
- There is no need to recompile the client kernel or server kernel if a class definition used as an interface for a remote RPC method call changes. It is often sufficient to update the tables with the new class definition.

5.3 Implementation details

The RPC system is built on top of the *Choices* message passing system[7]. The proxy object mechanism uses information generated by Proxify++ in the form of proxy object tables to implement method calls to the objects corresponding to the proxy. A proxy object table includes information describing the methods of an object, and is independent of whether the object is in the kernel or remote. A few extensions to the syntax of header files allows the specification of *read only* and *write only* parameters, array size, levels of pointer indirection, and null-terminated strings as parameters. This information is needed for marshalling arguments to remote methods. The *ObjectProxy* class is subclassed to *RemoteProxy* for remote access to objects and *KernelProxy* for local access to the kernel. All the methods of the parent class are redefined to use the appropriate kernel or remote method call interpretation of the proxy tables. The way in which a kernel object is accessed using a *KernelProxy* has already been discussed in Section 3.

Invoking methods on remote objects : A *RemoteProxy* object representing a remote object is allocated during *NameServer* lookup on that object. All method invocations on the remote object are handled by the *RemoteProxy*. When an application invokes the remote object's method, the corresponding *RemoteProxy* object method invokes a generic table interpreter function and passes a reference to the proxy table entry for the class and method. Each table entry stores information about the appropriate marshalling routines for each parameter in the remote method. The marshalling functions which are invoked for each parameter by the table interpreter are machine dependent. For example on stack based machines in a network, marshalling involves reading each parameter from the stack and packing it into the appropriate transmission format. Parameters of the same type share the same marshalling function. These functions also handle other machinene dependent translations like byte ordering. The *RemoteProxy* also implements parameter validation and communication with the server. The *RemoteProxy* object has a handle to the MessageContainer of its server. The *RemoteProxy* object also keeps specific information about the remote object and it satisfies some requests to the remote object locally. The arguments returned as a result from a method call are stored back using a similar approach.

Creating new remote objects : The new operator is used by applications to create new instances of any class. The result of a new operation is the creation of a new object in the local address space, in the kernel or in a remote address space. Local classes are handled by the language implementation and the handling of kernel classes is described by Dykstra[2]. Creating objects of remote classes is handled by the *ObjectServer* object. A new on a remote class is handled by invoking the construct method of the appropriate *ObjectServer*. The correct *ObjectServer* object is located by *NameServer* lookup. The construct routine is then invoked just like any other remote method invocation. This routine creates a new object in the correct domain, binds a server process to it if needed, registers the object with the *NameServer*, creates a *RemoteProxy* object for the remote object and returns it to the application. All remotely accessible objects have to be created using the *ObjectServer*, though once created, no further interaction with this object is required.

Creating servers : Services are exported by annotating class declarations with the proxiable keyword. A service class declares all methods that it wants to be globally accessible as proxiable. Instances of such a class can be created by applications if the class constructor is proxiable. If this is not the case, objects of this class can be created by the owner only. Any object of a service class is a server. All objects of

a service class are advertised in the *NameServer*. A server object is bound to a process which services all requests for method invocations on the object. A process may serve more than one object. Registration with the *NameServer* and binding a process is handled at object creation time by the *ObjectServer*.

6 Related work

Proxies were defined by Shapiro[10] as server representatives. He specified several desirable functions including access protocols, buffering, access control and communication functions. We use proxies as an interface to all objects not in the local address space. The use of proxies is transparent to the programmer. We also define some additional properties which might be useful, including failure handling, performance measurements and server versions.

Different mechanisms including ports, stubs, capabilities and virtual memory have been proposed as abstractions to simplify distributed programming and provide protected and transparent access to remote services[8, 3, 5]. Proxies subsume the functionality provided by ports, stubs and capabilities[10]. Comandos[5, 6] uses references to identify objects and uses a run time support library which is linked to user programs to perform object invocations. Specifically the *invoke* library function, which takes the method name as a parameter is used to implement a method call. Proxies are used in the special case of communicating with generic servers, while service requests to dedicated servers are handled by the system taking advantage of virtual memory. Our approach uses proxies for all non-local method invocations. Optimization of access depending on the location of the object, is handled by customizing the proxy. This provides a more uniform interface.

The remote procedure call (RPC) is a common abstraction used to provide easy and transparent access to services in a distributed system[4]. A common way of implementing RPC is by providing stubs for each of the remote functions in a library which is then linked with applications[11]. We use a table–driven approach to implement our method call facility which provides greater flexibility in customizing access and making changes to the server interface without sacrificing efficiency.

7 Conclusions

We have described the design of a table–driven proxy object RPC mechanism for *Choices* which provides controlled access to kernel and remote objects. The facility uses proxy objects to provide a transparent and reconfigurable application interface for a distributed system containing kernel and server service objects. The applications interface allows mobile services. It encourages microkernel experimentation since services can be relocatable transparently from the kernel to remote hosts without requiring applications to be recompiled or relinked. The scheme also permits proxy objects to perform local optimisations by, for example, caching and to support versioning of classes defining server objects by converting old method protocols to the new version of the protocol. We are examining the possibility of automating some optimisations and version support and adding some form of grouping or clustering to the *Choices* object model.

References

[1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[2] David W. Dykstra. *Object-Oriented Hierarchies Across Protection Boundaries*. PhD thesis, University of Illinois at Urbana-Champaign, April 1992.

[3] Richard Rashid et. al. Mach: A foundation for open systems. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 109–113, Pacific Grove, California, September 1989.

[4] Andrzej Goscinski. *Distributed Operating Systems The Logical Design*. Addison-Wesley, Reading, Massachusetts, 1991.

[5] Paulo Guedes and José Alves Marques. Operating system support for an object–oriented environment. In *Proceedings of the Second Workshop on Workstation Opeerating Systems*, pages 37–42, Pacific Grove, California, September 1989.

[6] Chris Horn. Is object orientation a good thing for distributed systems. In *Proceedings of the Euro-*

pean Workshop on Progress in Distributed Operating Systems and Distributed Systems Management, pages 60–74, Berlin, Germany, April 1989.

[7] Nayeem Islam and Roy H. Campbell. "Design Considerations for Shared Memory Multiprocessor Message Systems". In *IEEE Transactions on Parallel and Distributed Systems(to appear)*, October 1992.

[8] Sape J. Mullender, Guido van Rossum, Andrew S. Tannenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba a distributed operating system for the 1990s. *Computer*, 23(5):44–56, May 1990.

[9] Vincent F. Russo. *An Object-Oriented Operating System*. PhD thesis, University of Illinois at Urbana-Champaign, January 1991.

[10] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proc. 6th. Intl. Conf. on Distributed Computer Systems*, May 1986.

[11] B. H. Tay and A. L. Ananda. A Survey of Remote Procedure Calls. *ACM Operating Systems Review*, 24(3):68–79, July 1990.